Lab 3 Report

Evan Cole
School of Electrical and Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
evan.cole@okstate.edu

Landon Brown
School of Electrical and Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
landon.d.brown@okstate.edu

I. INTRODUCTION

Combinational logic is extremely important when it comes to the creation and implementation of digital logic, but there is only so much you can do with combinational logic alone. Sequential logic is a tool that lets us do different things at different times without having to hardcode something every single time the clock changes. One of the most important parts of sequential logic is known as a finite state machine, or an FSM. This logic is the primary coding that we used to execute this lab. For this lab we were given the project of blinking LEDs in a certain order like the tail lights for a thunderbird and a finite state machine is the one of the best ways of doing this.

II. BASELINE DESIGN

The base design of a FSM is quite simple as you have states and the input states are the determination for what the next output or state is. There are many different things you can do with a finite state machine, but most importantly is that you can represent one state at a time and use hardware and software to delay when it updates to the next state. This is a very important concept for lab 3.

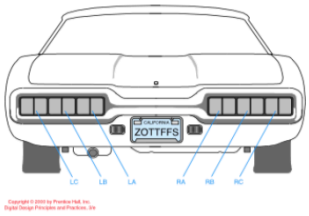


Figure 1: Thunderbird Tail Lights [1]

Fig. 1

The goal of lab three was to design a FSM that can represent the tail lights of a Thunderbird (Figure 1 from lab3 pdf).

III. DETAILED DESIGN

The FSM that we created to execute this lab used multiple different states and a couple inputs. Your first objective is trying to make a FSM diagram and a truth table of the problem at hand. Once this is done we can better conceptualize the task, and more easily implement code for it. The best place to start is with a diagram because that makes the truth table much easier to create. The diagram and table are shown below in figure 2 and 3 respectively.

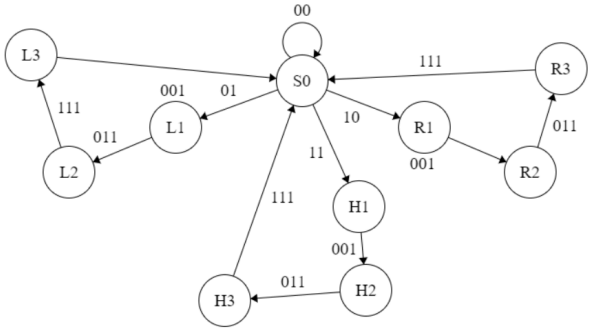


Fig. 2

Inputs	Current State	Next State	Output
00	S0	S0	000000
10	S0	R1	000000
10	R1	R2	000100
10	R2	R3	000110
10	R3	S0	000111
01	S0	L1	000000
01	L1	L2	001000
01	L2	L3	011000
01	L3	S0	111000
11	S0	H1	000000
11	H1	H2	001100
11	H2	H3	011110
11	H3	S0	111111

Fig. 3

When we have this we can move on to the Actual design of it and turn it into functional code. We start with FSM and use an input that tells your FSM what state your fsm will move to so for example if you bit a 01 it turns on the left side of the lights. While 10 turns the right side on and 11 for both you can detect this with an if else statement mixed in with your FSM.
When declaring your states you have to use typedef and declare it as logic like this

typedef enum logic [9:0] {S0, S1, S2, S3, R1, R2 ,R3, H1,H2,H3} statetype;

This declares all of your states that you need for the machine.

We decided to represent the taillights of a Thunderbird using a 6 bit number. The original number is 000000. The code then takes in an input, and based off of that input it sends it to one of the three different directions of the finite state machine. This can be seen in the diagram above. If the input is 01. Then the 6 bit number will be changed into a different 6 bit number in the next state, 001000. Then after some time it gets sent to the next state which is 011000. Then it gets sent to the last state of the "left side" which is 111000, and finally it gets reset back to 000000. This process repeats for as long as the input 01 is still getting put into the system. The output for each input is as follows:

01: 000000->001000->011000->111000->000000…
10: 000000->000100->000110->000111->000000…
11: 000000->001100->011110->111111->000000…

IV. TESTING STRATEGY

Once we got the FSM working it was time for the testing to see that everything was working as attended. For this we use model sim and a testbench. But before we load up modelsim we have to modify the testbench and do file. First thing to modify is we have to specify what time values the numbers need to turn on and off. The second thing is your Fsm.do file for the variables that you need to show up in the modelsim waveform. The last thing that you need to do is extend the time so you can see all the possible states for the FSM. When all that is done you can load it up and look at the waveforms. Figure 4 is a screenshot of the right blinker working as attended and with checking the other states and seeing they work as attended. It is ready for the FPGA.

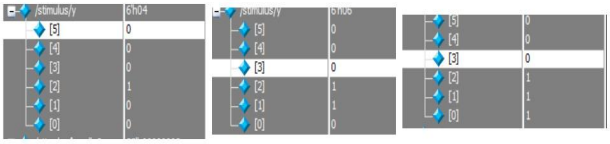


Fig. 4

V. EVALUATION

Before placing our code onto the FPGA we had to make sure it worked as intended in the vivado software as well. The vivado system is where we decide what our physical inputs should be and where we should send our outputs. We figured that the best way to do the inputs were on switches, and because we only had 3 different 2 bit inputs, 00, 01, and 11, we only needed to implement two of the switches. After this we decided that the best place to put our output would be on 6 led lights that simply lit up when the corresponding bit of our original 6 bit number went hot. This made it look just like the taillights of a thunderbird, and we could tell easily if there were errors in our program because the lights would either work correctly or something would be wrong. After implementing all of this into vivado we were finally ready to put it on the board. After doing that we relised it moved way to fast for us be able to see it at correct speeds so we

implemented a clock divider to slow it down for us. The output of the hazard light input is shown in figure 5 below.

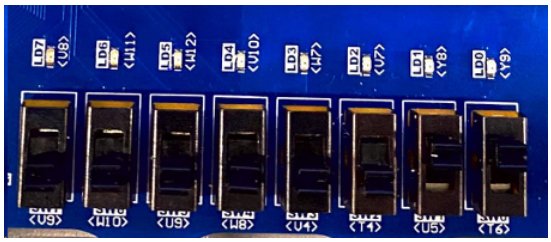
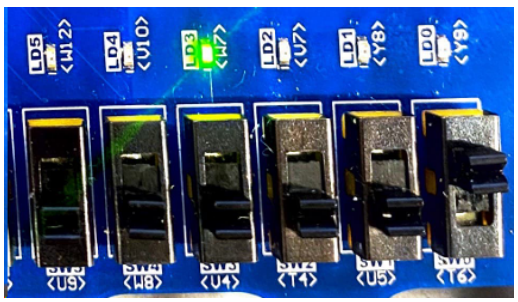
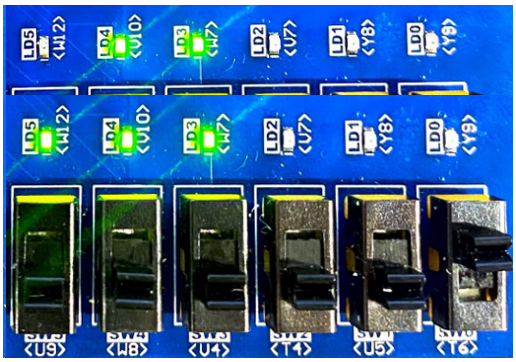


Fig. 5

This shows that the code correctly followed our FSM diagram. The 6 bit number went from 000000 in the first picture to 001100 in the second to 011110 in the third then finally to 111111. This then repeated for as long as the switches still showed 11. The output for the left input is shown below.





This also shows a correct representation of our FSM as does the input for the right side. Although when we first implemented it onto the board it was moving much too fast so we had to delay the clock speed a little bit to make sure we could actually see what was going on. After looking at the PPA files, this shows it takes about as much power as a GPU in a computer. And with all that done we can see the final schematic in fig 6 of the FSM.

Fig. 6